Vanshika Ambwani
D20A-2

BLOCKCHAIN LAB EXP-1

Aim-
Write a Python program to understand SHA and Cryptography in Blockchain, Merkle root tree hash

Theory-

1. Cryptographic Hash Functions in Blockchain

Cryptographic hash functions are algorithms that take input data of any size and generate a fixed-length, irreversible hash. In blockchain, SHA-256 is commonly used to secure transactions and blocks. These functions are deterministic, collision-resistant, and produce a completely different output even with a small change in input. Hash functions ensure **data integrity**, link blocks together securely, and prevent tampering, forming the backbone of blockchain security.

2. Merkle Tree

A Merkle Tree is a binary tree structure that efficiently summarizes and verifies large amounts of data using cryptographic hashes. In a Merkle Tree, the **leaf nodes** represent the hashes of individual transactions, while the **parent nodes** are hashes of their child nodes combined. The topmost node, called the **Merkle Root**, uniquely represents all the transactions in the block. This structure is widely used in blockchain to secure transaction integrity.

3. Structure of Merkle Tree

- **Leaf Nodes:** Hash of individual transactions
- **Intermediate Nodes:** Hash of two child nodes combined
- **Merkle Root:** Final single hash at the top

If the number of nodes is odd, the **last hash is duplicated**.

4. Merkle Rule

The Merkle rule states that each parent node in the tree is computed as the hash of its two child nodes. If there is an odd number of nodes at any level, the last node is duplicated before hashing. This process continues recursively until a single hash, called the Merkle Root, is obtained. This rule ensures that the root hash uniquely represents all underlying transactions and any modification in a transaction can be quickly detected.

5. Working of Merkle Tree

The working of a Merkle Tree involves first hashing all individual transactions to create leaf nodes. Then, pairs of hashes are combined and hashed repeatedly to form intermediate nodes. This process continues until only the Merkle Root remains at the top of the tree. The Merkle Root serves as a single, verifiable summary of all transactions, allowing efficient verification without needing to examine every transaction individually.

6. Benefits of Merkle Tree

- Efficient data verification
- Reduced storage requirements
- Fast integrity checking
- Tamper detection
- Scalable for large datasets

7. Use of Merkle Tree in Blockchain

In blockchain, Merkle Trees are used to securely store transactions in each block and generate the Merkle Root. This allows nodes to verify transactions efficiently without downloading the entire block, which is especially useful for lightweight clients. Merkle Trees also ensure data integrity, prevent tampering, and maintain a verifiable link between all transactions within a block

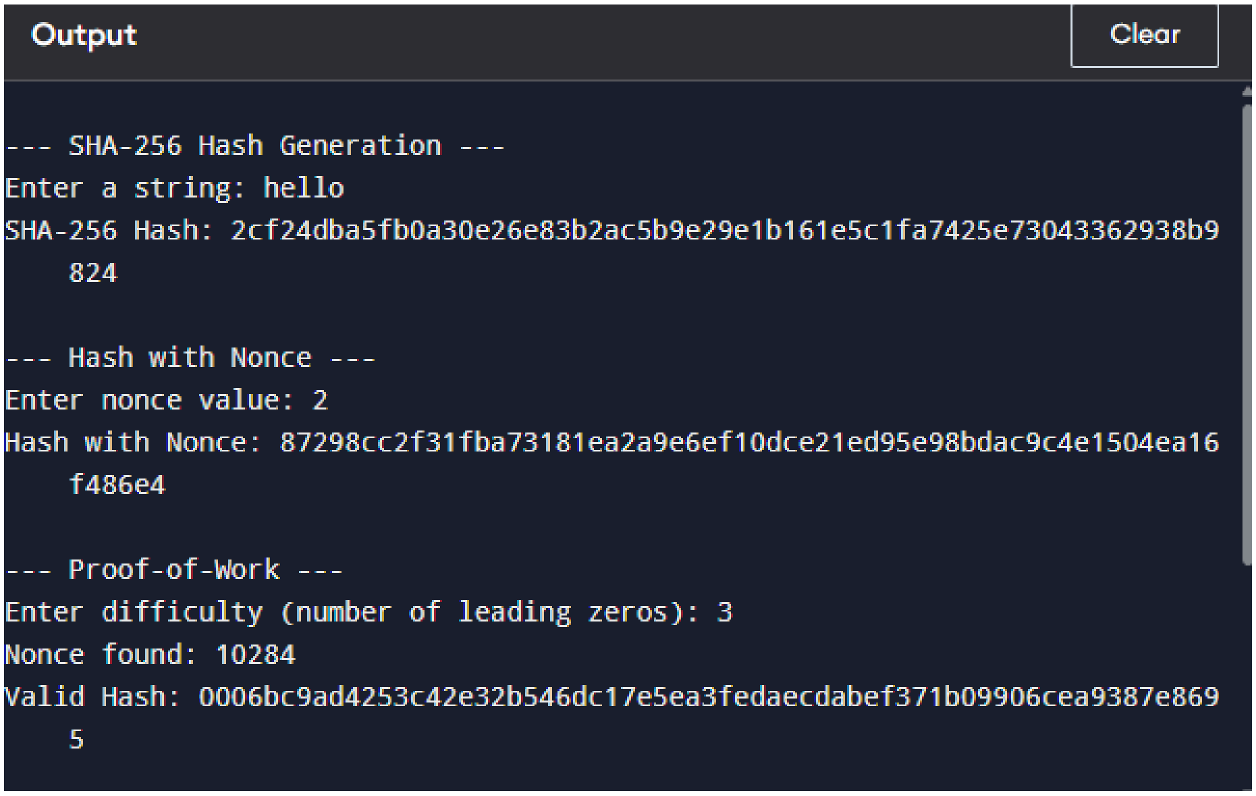
8. Use Cases of Merkle Tree

- Blockchain transaction verification
- Distributed systems
- File integrity verification
- Version control systems (Git)
- Peer-to-peer networks
- Database consistency checking

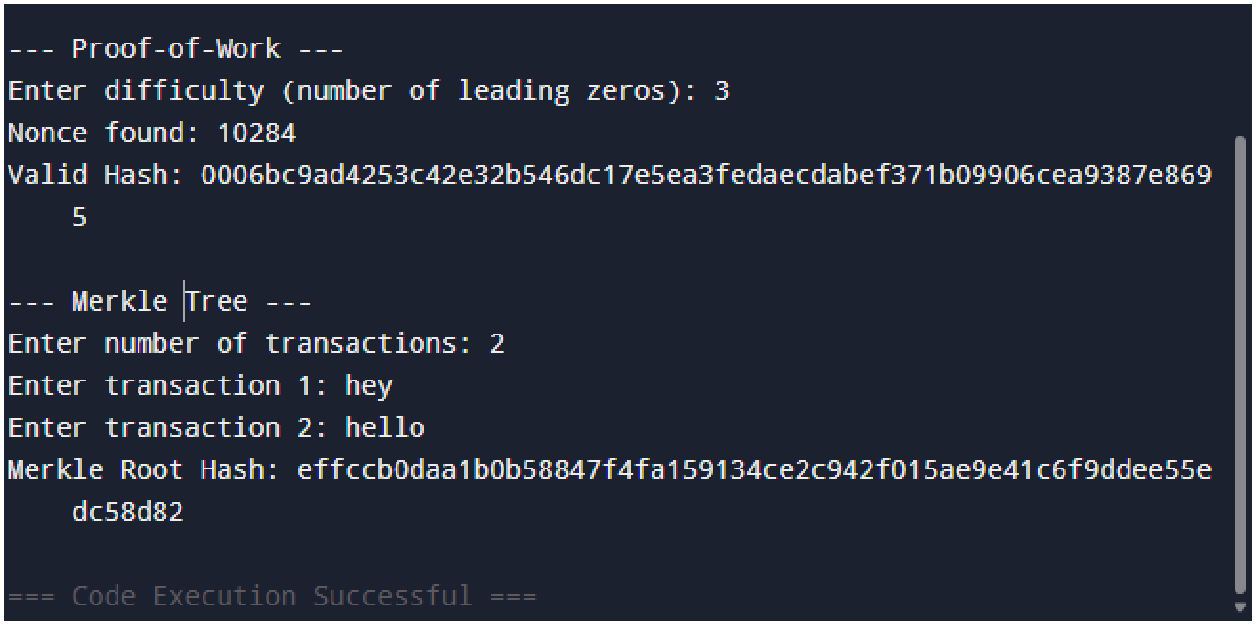
Code-

```
import hashlib

# -----------------------------------------
# 1. SHA-256 Hash Generation
# -----------------------------------------
def sha256_hash(data):
    return hashlib.sha256(data.encode()).hexdigest()


# -----------------------------------------
# 2. Hash Generation with Nonce
# -----------------------------------------
def hash_with_nonce(data, nonce):
    combined = data + str(nonce)
    return sha256_hash(combined)


# -----------------------------------------
# 3. Proof-of-Work (Mining Simulation)
# -----------------------------------------
def proof_of_work(data, difficulty):
    nonce = 0
    target = '0' * difficulty

    while True:
        hash_result = hash_with_nonce(data, nonce)
        if hash_result.startswith(target):
            return nonce, hash_result
        nonce += 1


# -----------------------------------------
# 4. Merkle Tree Construction
# -----------------------------------------
def merkle_root(transactions):
    # Generate initial transaction hashes
    hashes = [sha256_hash(tx) for tx in transactions]

    # Continue hashing until only one hash remains
    while len(hashes) > 1:
        if len(hashes) % 2 != 0:
            hashes.append(hashes[-1])  # Duplicate last hash if odd
        new_level = []
        for i in range(0, len(hashes), 2):
            combined_hash = hashes[i] + hashes[i + 1]
            new_level.append(sha256_hash(combined_hash))

        hashes = new_level

    return hashes[0]


# -----------------------------------------
# MAIN PROGRAM
# -----------------------------------------
if __name__ == "__main__":
    print("\n--- SHA-256 Hash Generation ---")
    message = input("Enter a string: ")
    print("SHA-256 Hash:", sha256_hash(message))

    print("\n--- Hash with Nonce ---")
    nonce = int(input("Enter nonce value: "))
    print("Hash with Nonce:", hash_with_nonce(message, nonce))

    print("\n--- Proof-of-Work ---")
    difficulty = int(input("Enter difficulty (number of leading zeros): "))
    nonce, pow_hash = proof_of_work(message, difficulty)
    print("Nonce found:", nonce)
    print("Valid Hash:", pow_hash)

    print("\n--- Merkle Tree ---")
    n = int(input("Enter number of transactions: "))
    transactions = []
    for i in range(n):
        tx = input(f"Enter transaction {i+1}: ")
        transactions.append(tx)

    root = merkle_root(transactions)
    print("Merkle Root Hash:", root)
```

Output-



```
--- SHA-256 Hash Generation ---
Enter a string: hello
SHA-256 Hash: 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9
    824

--- Hash with Nonce ---
Enter nonce value: 2
Hash with Nonce: 87298cc2f31fba73181ea2a9e6ef10dce21ed95e98bdac9c4e1504ea16
    f486e4

--- Proof-of-Work ---
Enter difficulty (number of leading zeros): 3
Nonce found: 10284
Valid Hash: 0006bc9ad4253c42e32b546dc17e5ea3fedaecdabef371b09906cea9387e869
    5
```



```
--- Proof-of-Work ---
Enter difficulty (number of leading zeros): 3
Nonce found: 10284
Valid Hash: 0006bc9ad4253c42e32b546dc17e5ea3fedaecdabef371b09906cea9387e869
    5

--- Merkle Tree ---
Enter number of transactions: 2
Enter transaction 1: hey
Enter transaction 2: hello
Merkle Root Hash: effccb0daa1b0b58847f4fa159134ce2c942f015ae9e41c6f9ddee55e
    dc58d82

=== Code Execution Successful ===
```